Power Awareness through Selective Dynamically Optimized Traces

Roni Rosner, Yoav Almog, Micha Moffie, Naftali Schwartz and Avi Mendelson

Microprocessor Research
Intel Labs, Haifa, Israel
{roni.rosner, yoav.almog, micha.moffie, naftali.schwartz, avi.mendelson} @ intel.com

Abstract

We present the PARROT concept that seeks to achieve higher performance with reduced energy consumption through gradual optimization of frequently executed code traces. The PARROT microarchitectural framework integrates trace caching, dynamic optimizations and pipeline decoupling. We employ a selective approach for applying complex mechanisms only upon the most frequently used traces to maximize the performance gain at any given power constraint, thus attaining finer control of tradeoffs between performance and power awareness.

We show that the PARROT based microarchitecture can improve the performance of aggressively designed processors by providing the means to improve the utilization of their more elaborate resources. At the same time, rigorous selection of traces prior to storage and optimization provides the key to attenuating increases in the power budget.

For resource-constrained designs, PARROT based architectures deliver better performance (up to an average 16% increase in IPC) at a comparable energy level, whereas the conventional path to a similar performance improvement consumes an average 70% more energy. Meanwhile, for those designs which can tolerate a higher power budget, PARROT gracefully scales up to use additional execution resources in a uniformly efficient manner. In particular, a PARROT-style doubly-wide machine delivers an average 45% IPC improvement while actually improving the cubic-MIPS-per-WATT power awareness metric by over 50%.

1. Introduction

Revolutionary and evolutionary advances in microarchitecture and process technology have sustained Moore's law—doubling both transistor density and processor performance on average every 18 months—against all odds. However, despite the decreasing power consumption per individual transistor characteristic of newer process technologies, innovations in microarchitectural techniques and process technology increases the total number of transistors and their operation frequency, resulting in an overall increase in power consumption as well as their power density. Doubling cache sizes, adding new and more complex types of speculation and other modern innovations require so many more resources that their own benefit is compromised. Current processors have become power limited, that is, they can only operate at limited frequency, preventing them from achieving their full microarchitectural performance potential.

In this paper we focus on power-aware microarchitectures for high-performance, general-purpose processors. The essential challenge within this domain is the increasingly poor scaling of performance with power consumption. Nevertheless, the concepts and methods we present can be applied to special purpose, low-power and low-performance processor types as well.

Traditionally, processor microarchitecture can be divided into front-end and back-end parts. The front-end supplies the instructions to be executed in program order and the back-end executes them, possibly out-of-order, and commits the executed instructions (in-order again) updating the architectural state. The front-end bandwidth is crucial for the overall performance of the system — it depends on the number of instructions that can be predicted, fetched, decoded and renamed in parallel. Since decoding variable-length CISC instructions, such as those characteristic of the IA32 architecture, is an essentially serial activity, huge decoders must be employed to introduce the needed parallelism. Such decoders consume vast quantities of power; incremental enlargement brings diminishing incremental performance returns.

The back-end presents different power/performance tradeoffs. Although execution bandwidth scales with the number of execution units, instruction scheduling efficacy does not linearly scale with instruction window width. The power and complexity of dynamic scheduling depends on execution bandwidth as well as on program behavior and the instruction window size [18][3]. This is one of the main reasons that low-power architectures tend to spurn dynamic scheduling, preferring instead (VLIW like) static scheduling as a means for reducing the scheduling power [24].

This paper presents a different approach for microprocessor design that addresses various challenges posed by both the fetch and scheduling stages. Following Amdhal's Law, we propose to take advantage of the time-honored hot/cold (or 90/10) paradigm. The hot/cold paradigm asserts that a "small" portion of the ***static*** program code is responsible for "most" of its ***dynamic*** execution. This paradigm is utilized in the context of profiling compilers, dynamic translators and other program-manipulating systems in which most optimization effort is targeted at the small portion of ***frequent*** (or ***hot***) program code. PARROT applies similar principles at the hardware level and proposes to build the entire micro-architecture according to this paradigm.

Identifying frequently executed code sections for optimization has been applied in the software-based schemes reported in [19][9][1][13]. More recently, similar methods were suggested for hardware-based systems [21][22][26][25][10][29][33]. The various proposals differ in the methodology and resources used for detecting the hot paths, the structure and address space used for storing them, and the timing and resources used for optimization.

The PARROT concept aims to aggressively exploit the hot/cold paradigm in hardware for the benefit of both processor performance and power awareness, as indicated by the PARROT acronym: Power-Aware aRchitecture Running Optimized Traces. The current study examines several microarchitectural alternatives based on this concept (we refer to them as PARROT microarchitectures). These microarchitectures are organized around an optimized trace cache. Trace caches [27][31] were mainly proposed for obtaining higher fetch bandwidth by capturing and reusing the dynamic flow of instructions irrespective of program order [23][28]. In [29] it was observed that trace-cache based mechanisms are also useful for reducing power consumption, but that differing characteristics of traces in a system may enhance either power or performance. In the current work we study the limits of simultaneous achievement of both performance and power awareness.

The PARROT microarchitecture is designed to effectively identify the most frequent sequences of program code, aggressively optimize them once, and then efficiently execute them many times. Trace selection and filtering identify the hot code, a dynamic optimizer optimizes it, and a trace cache stores traces for repeated execution. Gradual construction of traces, pipeline decoupling, and specific trace optimizations are key factors for power awareness. Limiting the hardware dedicated to the cold part of the code may exact a small price in performance. In return, more aggressive hardware may be used to improve performance/power tradeoffs for the dominant hot segments of the code. Indeed, our results show that no additional amortized energy is spent for the optimization of hot traces.

The simulation framework provides both performance and energy measurements in order to establish performance and power/performance tradeoffs. In order to compare the new PARROT based microarchitecture to "conventional" high performance processors, we consider two "reference" microarchitectural models: a standard modern processor (a 4-wide, super-scalar, super-pipelined, out-of-order microarchitecture) and its straightforward 8-wide extension. Within the PARROT framework, we examine several new power-aware high-performance alternatives that include trace-cache extensions of the reference models and dynamic hot-trace optimization. Our results show that PARROT microarchitectures indeed attain both high performance and power awareness by efficiently exploiting the machine's available resources.

PARROT's uniqueness is in the application of decoupling and dynamic optimization techniques to achieve better performance with less energy consumption. Similar ideas, mainly targeting performance issues, have been suggested before. The rePlay system [25], in particular, has much in common with PARROT techniques. PARROT and rePlay share the dual front-end, the decoupled, post-retirement construction of traces, and dynamic optimization of traces stored in a trace-cache. To promote power awareness, PARROT proposes a finer decoupling of trace construction based on gradual filtering in order to improve controllability of competing design metrics. PARROT's trace construction criteria are mostly static, enabling better adaptability to program structure. A good example is the handling of loops: by cutting loops at iteration boundaries, the PARROT microarchitecture prevents redundancy in the trace cache while still allowing loop unrolling. In contrast, the dynamic selection criteria of rePlay are in better synergy with the prediction mechanism.

Our results complement and strengthen the rePlay study [33] showing the significant contribution of dynamic optimizations to IA32-based processors. However, PARROT goes beyond rePlay optimization scope by introducing core-specific optimizations which heavily exploit the fact that the optimizations are integrated into the hardware (their particular contribution is quantified in a separate paper [1]).

The dichotomy between regular and irregular code led Turboscalar [4] into the design of separate pipelines, a deep-and-narrow pipeline for the irregular code, and a shallow-and-wide pipeline for regular code (an alternative, power-oriented dichotomy is proposed in [7]). Turboscalar execution optimizes the resource bandwidth required for higher performance. PARROT builds upon a similar conceptual separation of pipelines but allows for

alternative implementations of the concept. The PARROT framework is used to study performance and energy trade-offs of several alternative structures and organization of the processor pipeline over a wide range of benchmarks.

The DAISY system [9] is designed to emulate multiple ISAs on a simple VLIW architecture. It employs a software layer responsible for compatible translation of "legacy" code into VLIW format. The DAISY approach allows for cheap and low-power implementation that can deliver high ILP for the portions of legacy code suitable for VLIW execution. In like fashion to other non-native binary translators, all code must be transformed to DAISY's ISA, necessitating machine cycles for translating non-frequent and non-optimizable code as well. Furthermore, code that does not lend itself to efficient VLIW execution (e.g., code suffering from L1-data misses) may suffer from severe performance degradation.

The rest of the paper is organized as follows. Section 2 describes the PARROT concept and microarchitecture in detail. Section 3 describes the simulation framework and defines the microarchitectural models compared in the current study. Section 4 presents the simulation results, and finally, Section 5 concludes with a summary and ideas for future studies.

2. PARROT Microarchitectural Framework

In this section we study various microarchitectures based on the PARROT concept. Performance and energy are evaluated within the simulation framework presented in Section 3.

2.1. The PARROT Concept

PARROT is based on the following observations:

- The working set of a program at any given time is relatively small.
- Much of the complexity excesses of modern OOO processors result from handling rare "corner cases".
- Small segments of code which are repeatedly executed ("hot-traces") typically cover most of the program's working set at any given time.
- The hot segments of code behave differently than the rest of the code, namely they are more regular and predictable, and consequently they exhibit higher potential for ILP extraction than the other, less frequently executed parts of the code.

The PARROT concept suggests basing the development of high performance power-aware system on an ***asymmetric decoupling*** of the processor pipelines; a slightly different decoupling concept is proposed in [4]. Figure 2.1 presents the conceptual structure of a decoupled PARROT system. The left-hand part is responsible for executing the cold portion of the code and the right-hand side executes the hot portion of the code.

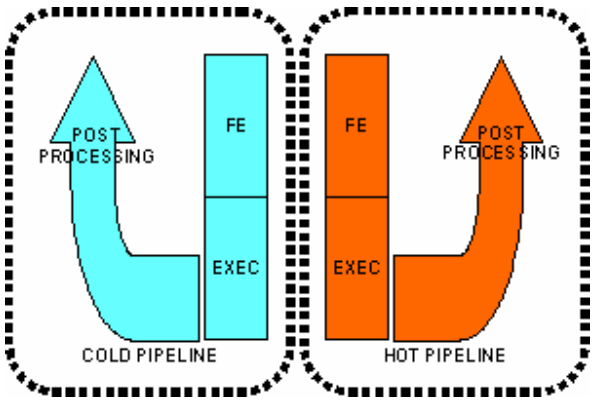

Figure 2.1 Schematic PARROT split-core μarch

The cold and hot subsystems have a similar high level structure, with each being comprised of foreground and background components operating in parallel. The foreground components include the front-end and execution pipelines which are tuned for either cold or hot portions of the code. The background components post-process the instruction flow out of the foreground pipeline making "off critical path" decisions such as when to move from the cold subsystem to the hot subsystem and when to apply further optimizations. Synchronization elements are required for arbitrating and switching states between the pipelines and for preserving global program order (architectural state).

Previous research [29][17] indicates that a trace cache can be very efficient in handling hot code, provided this code has been sufficiently well identified. (This is especially true in Intel's IA32 ISA which features variable length instructions.) Thus, we base the cold subsystem on instructions fetched from an instruction cache whereas the hot subsystem is based on traces fetched from a trace cache. Both power-awareness and trace-cache effectiveness considerations limit trace construction and trace-cache insertion to frequently executed code sections. Furthermore, PARROT ***gradually*** applies dynamic optimizations — the hotter the trace is, the more aggressive power aware optimizations are applied.

Reusability of hardware work and results is important for both performance and energy savings. In the PARROT framework, the trace-cache stores ***decoded*** traces and is thus a container for reuse of decoding results. It also stores optimized traces allowing for multiple reuses of trace optimizations.

Dynamic optimizations have several advantages. Dynamic information available at optimization time, most notably control resolution (outcome of trace internal branches), enables optimizations that are impossible for a static compiler. Decoupling these optimizations from the

foreground pipeline allows for more aggressive optimizations than on-the-fly optimizations that can be performed within a standard execution pipeline. In order to take full advantage of such a relaxed hardware context, ***atomicity*** of trace is assumed. Trace semantics that assumes atomic commit of traces permits for very aggressive optimizations across basic-block boundaries, including elimination and reordering of instructions, provided the overall semantics of the trace is preserved

Another advantage of micro-architectural level optimizations is that of ***architectural transparency***. The hardware is capable of optimizing legacy code, exploiting new microarchitectural features without the need for recompilation.

2.2. Traces and Trace-Selection

An execution ***trace*** is a sequence of operations representing a continuous segment of the dynamic flow (execution) of a program. Traces may contain execution beyond control-transfer instructions (CTIs), and so a trace may extend over several basic blocks.

In the current study we consider ***decoded atomic traces***. These traces contain ***decoded*** micro-operations (uops) and enable reuse of decode activity, thus saving energy [32][29] (decoded traces are of special value for IA32). Traces are constructed from the originally decoded uops in program order, but may later be optimized, resulting in a different, reordered, generally shorter sequence of uops.

Atomic traces are single-entry and single-exit [20][31]. Although atomic trace retirement requires a relatively complicated recovery mechanism and longer recovery time for the case of misprediction, it enables more aggressive optimizations, including uop reordering and elimination and branch promotion [25][26] and may efficiently utilize advanced trace prediction techniques [15].

Trace selection is the process of constructing particular traces out of a dynamic sequence of instructions. This process may be ***deterministic*** if applied to the fully predictable sequence of in-order committed instructions, or ***speculative*** if applied to any previous stage in the pipeline in which instruction are potentially mispredicted. A comprehensive study of trace-selection techniques can be found in [30]. In the current study we apply the following deterministic selection criteria:

- Capacity limitation: traces are constructed into frames of at most 64 uops.
- Complete ***basic-blocks***: with the exception of extremely large basic blocks, traces always terminate on CTIs.
- Terminating CTIs: all ***indirect jumps*** and software exceptions terminate basic-blocks, except RETURN instructions. In addition, ***backward taken*** branches terminate a trace.
- RETURN instructions terminate traces only if they exit the outermost procedure context already encountered in the current trace. This condition is verified using a context counter which is incremented on procedure calls and decremented on procedure returns. This criterion achieves the effects of ***procedure inlining***.
- If two or more consecutive traces are identical, they are joined into a single trace, until the capacity limit is reached. This criterion, together with the taken-backwards termination condition on traces, achieves the effects of ***explicit loop unrolling***, an enabler for other optimizations.

With these criteria, unique ***trace identifiers*** (**TIDs**) can be compacted into a single address and a sequence of branch directions (taken/not taken). The only indirect CTI in this construction is a RETURN, but since its context is available inside the trace, its target address is implicitly available.

2.3. Microarchitecture

A ***split-execution*** implementation, faithful to the PARROT microarchitectural concept, consists of two disjoint subsystems for the cold and for the hot paths as presented in Figure 2.1. Consequently, different execution engines can be employed by each subsystem. For example, a wider execution engine could be used for the higher-bandwidth, trace-based hot subsystem. More sophisticated variants, not considered in the current study, may employ completely different execution models for the disjoint subsystems, such as in-order and out-of-order. An optimized ***unified-execution*** implementation is presented in Figure 2.2. It duplicates the front-end but shares the execution resources between the hot and cold subsystems.

The following description generally applies to both the split and unified-execution microarchitecture implementations. A schematic description of the major components is depicted in Figure 2.3. Both cold and hot pipelines operate in two phases: the ***foreground phase*** which is responsible for the fetch-to-execution pipeline; the ***background*** (or ***post-processing***) ***phase*** serves for selecting frequent parts of the just executed code, improving (optimizing) them and potentially promoting them to a "hotter" level.

More specifically, the background phase of the cold subsystem identifies frequent IA32 instruction sequences and captures them as traces in the trace cache. It is composed of TID selection, TID hot-filtering and finally trace-construction and insertion into the trace-cache. Since all committed instructions go into the TID selection phase, continuous training of both trace predictor and hot filter is assured. Nevertheless, only those TIDs that pass the hot-filter continue to the trace construction stage. The background phase of the hot subsystem identifies the most frequent (blazing) traces, optimizes them and finally

inserts them back into the trace cache. As noted above, post-processing is performed gradually, so the longer a trace is used the more aggressive optimizations are applied to it.

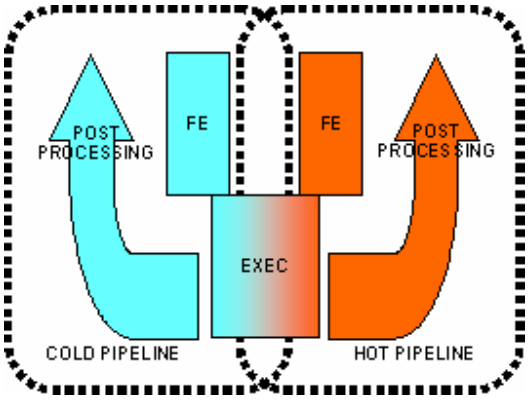

Figure 2.2 Schematic PARROT unified-core µarch

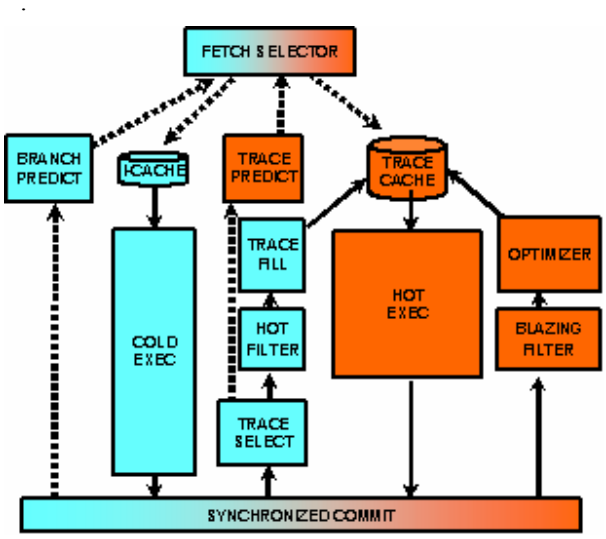

Figure 2.3 PARROT main µarch components

Two predictors are employed: A branch predictor predicts the next cache line designated to be fetched from the instruction cache for execution on the cold pipeline. A trace predictor predicts the TID of the next trace designated to be fetched from the trace cache and executed on the hot pipeline. Each predictor is based on a global history register (GHR). The GHR is updated for each CTI being executed. Both predictors support speculative update upon fetch and real update upon commit. It is important to note the asymmetry between the two mechanisms: while the cold pipeline always has a "default" fall-through prediction, and every legal prediction could in principle be fetched from the instruction cache, the trace predictor can make relevant predictions only after proper training, and a predicted TID may reflect a trace that is not present in the trace cache.

The fetch-selector chooses between the execution pipelines by consulting both the lower priority branch predictor and the higher-priority trace predictor. Whenever the trace-predictor is successful in making a next TID prediction, the hot pipeline is selected for executing the predicted trace. The TID is looked up in the trace cache, and if the trace is successfully fetched it is executed on the hot pipeline. All other cases result in cold pipeline execution directed by the branch predictor.

In the foreground phase the pipelines are executing sequences of uops originating from either (cold) instructions or (hot) traces. This is performed by either a pair of cold and hot cores working in tandem, or by a single unified core. A split core design enables core specialization: the cold core may focus on the execution of rare but complex operations or be less performance aggressive, while the hot core may excel at aggressive execution of atomic traces, employ simplified renaming schemes or rely on dynamic scheduling performed by the optimizer. On the other hand, the split core microarchitecture increases die size and introduces complexities associated with cold/hot state switches. Compared to the split core, a unified core simplifies the design, and reduces both die size and idle power. In this study we consider standard super-scalar out-of-order cores only, in both split and unified configurations.

For the split core design, the state switch mechanism ensures that values computed and stored in the register file of one core are used at the appropriate time and place in the second core. This is accomplished by tracking for each register the last writer uop of the code preceding the switch and the first reader uop of the code following the switch, and assuring that the reader uop is not permitted to execute until the writer uop has undergone writeback and the value has been communicated to the second core. This mechanism is similar to the forwarding of register values used to satisfy data dependences across stages under the Expandable Split Window paradigm [11].

The commit stage is responsible for committing IA32 instructions to the architectural state. This stage deals with two synchronization issues: First, it has to commit instructions in program order, which means that in a split microarchitecture instructions must contain markers which permit reconstruction of the global program order from the interleaved execution. Secondly, the atomic hot traces should be committed at once as a single entity, requiring a mechanism for state accumulation. Note that the atomic commit we consider is conceptual. Individual uops of a trace are retired at a rate commensurate with the overall machine width. For such a gradual scheme, only moderate enlargement of non-critical machine resources is necessary. Upon any intermediate event that prevents the full completion of the trace, the remaining uops and accumulated state are flushed, and the architectural state

at the commencement of the trace is restored. Such distracting events may result from exceptions in the execution of the trace itself, from failed **assert** uops (indicating trace mispredictions) or from external interrupts.

For post-processing cold code, PARROT employs a deterministic TID/trace build scheme. Uops originating from cold committed instructions are collected as long as all encountered CTIs satisfy the trace selection criteria (see Section 2.2). When a termination condition is reached, a new TID, generated from the collected CTIs, is used to train the trace predictor. If the TID is subsequently identified as frequent, the collected uops are used to construct an executable trace that can be inserted into the trace cache.

In order to identify the frequently executed instruction sequences, PARROT gradually employs two filtering mechanism: the hot filter, which is used for selecting frequent TIDs from among those constructed on the cold pipeline, and the blazing filter, which is used for selecting the most frequent TIDs from among those executed on the hot pipeline. Both filters are small caches that maintain access counters for each TID. Each trace execution increments the corresponding counter. Once the hot filter threshold is reached, the trace is constructed and inserted into the trace cache. When the blazing filter threshold is reached, the executed trace is optimized and written back to the trace cache, replacing the original.

PARROT employs dynamic optimizations on blazing traces (identified by the blazing filter) facilitating supreme efficiency in performance as well as power. The optimizer relies heavily on the atomicity assumption (manifested by **assert** operations [25]), which enables aggressive trace-transformations (e.g., reordering).

2.4. Dynamic Optimizations

The optimizations can be classified as either general purpose or core-specific. General purpose optimizations are independent of the underlying execution core. They include logic simplifications, constant propagation and dead code elimination. Core-specific optimizations include functional transformations such as micro-operation fusion and SIMDification, and global transformations such as partial renaming and dynamic critical path based scheduling.

Optimizations may result in: uop reduction, dependency elimination, simplified renaming and improved scheduling. Some optimizations, such as virtual renaming, contribute mainly to power/energy saving. Others, such as dependency elimination are performance oriented. Reducing the number of micro-operations contributes in general to both performance and energy savings.

A companion paper [1] is dedicated to a detailed study of dynamic optimizations in the context of PARROT. A breakdown of the impact of different classes of optimizations is used there to demonstrate the significance of core-specific optimizations on top of generic ones. The intimate relation with the execution hardware enables aggressive optimizations that more than double both the performance improvements and energy savings obtained compared with generic optimizations alone. In addition, a sensitivity study demonstrates that a relaxed design could be employed for such an aggressive optimizer due to the high reuse ratio for optimized traces obtained by virtue of the relatively high blazing threshold.

3. Simulation Framework

We employ an in-house proprietary performance and power simulation environment as a modeling and research vehicle for the PARROT microarchitecture. The simulation environment is designed with maximum flexibility and configurability in order to enable comparative study of the diverse set of microarchitectural alternatives described in Section 3.3 below, based on a diverse set of benchmark applications.

The simulators are trace-driven, simulating execution traces (not to be confused with microarchitectural traces) of applications compiled for the IA32 architecture. They implement all the components of the PARROT microarchitecture, including the less traditional optimizer which implements all of the optimizations and pre-computations described above.

3.1. Performance Simulation

The performance simulator incorporates a full memory hierarchy and newly designed components for the post-processing phases.

The software architecture includes a generic, highly configurable object-oriented execution core class which can be instantiated with a variable number of execution cores of widely differing characteristics, including machine width, number of ROB ports and number and latencies of execution units. Because it incorporates base class suitable for executing both cold-instructions and hot-traces, it can be used to construct widely differing machine configurations.

One distinctive feature of our simulation framework is the ***abstract instruction***. An abstract instruction can be defined as a "committable work unit" and so has a different interpretation within the cold and hot pipelines. Within the cold pipeline, it remains the familiar instruction, but within the hot pipeline, since an optimized trace is committed atomically (either the entire trace is committed or none of its instructions is committed) we consider the trace to be an "abstract instruction". This design decision enabled both hot and cold execution cores to be instantiations of a generic code base.

Furthermore, to maintain a high-level plug-and-play simulation semantics, the execution cluster is not a transformer of instructions to uops, as is customary. Rather, it outputs the same abstract instruction data type that it accepts for input. This design decision greatly increased the versatility of the entire system and allowed experimentation with interesting configurations which short-circuited one or both execution clusters.

We modeled the optimizer as a non-pipelined unit which stores one trace in a simplified ROB-like structure and analyzes the constituent uops sequentially, employing standard rename tables. The optimizer maintains a static dependency graph, which is used across different optimization passes. The optimizations are carried out in several passes, each pass responsible for optimizations that require similar resources. Consequently, we have modeled a significant delay (on the order of 100 cycles) for optimization of a single trace. More realistic schemes are indeed possible, and will be reported on in future work.

3.2. Energy Simulation

For energy simulation we use proprietary tools which are based on a combination of the WATTCH-like [6] and TEM^{2}P^{2}EST-like [8] approaches. The use of proprietary tools enabled us to validate PARROT models against up-to-date data relevant to modern technologies in use at Intel. Nevertheless, we provide adequate details so that one can generate similar results using a comparable toolset.

Similar to WATTCH, we attach a "power tag" for every operation in the simulator, such as read from cache, write to register file, etc., and use the performance simulation to count how many times each event occurs during the execution. For all the microarchitectural components we inherit from existing architectures, we use accurate data extracted from real processors, modulo normalization for the process technology. For new elements in the microarchitecture, we use a method based on [8]. We start with formulas for small functional blocks in different styles and correlate each of them to the corresponding hardware implemented on recent technology. Then, we compose these building blocks into higher-level units. The power consumption of the "building blocks" of the model was taken from instances of buffers and tables from the original ROB, rename and scheduling stages of the pipeline, scaled appropriately, and used together with some combination of significant simulation events mapped to the original execution events.

After creating a power consumption matrix for each operation on each hardware unit, we use the performance simulation runs to provide the active power of each of the units/operations. We then use these parameters to compute total energy consumption for the entire model as well as to measure the relative contribution per unit as a fraction of the total energy consumption.

Leakage is derived from the dynamic power under some simplifying assumptions. We assume uniform leakage 1) in space over two coarse component types, the processor core and the level-2 cache, and 2) in time, modeling a consistently high temperature.

To emulate the high temperature, we choose the application with highest average dynamic-power *PMAX* of the base OOO model. This turns out to be swim of the SpecFP suite (see below). For a component area *A*, we define the uniform leakage power as *A* * *PMAX* * *T*, where *T* is a technology constant. We use technology constants of 5% for each MByte of level 2 cache and 40% for the standard core. These fairly large constants are used in accordance with the technological trend of increasing leakage. Thus, for a model with *M* Mbytes of L2 cache and a core of area *K* times the standard OOO core, the total leakage energy *LE* of an application running for *CYC* cycles is modeled by the formula

LE = PMAX * (0.05 * M + 0.4 * K) * CYC

This infrastructure produces energy estimations for the different micro-architecture models and their components, usable for global trade-off analysis.

3.3. Models

We modeled a variety of configurations to elucidate various aspects of the power and performance of the PARROT microarchitecture. The reference model (**N**) was configured to resemble a standard 4-wide OOO machine. In the ensuing discussion, *narrow* will refer to different variants of this standard 4-wide, while *wide* should be understood to refer to variants of a more generous 8-wide machine. The configuration space we consider is depicted in Table 3.1.

Building on our reference base narrow configuration, we created a theoretical configuration where all stages from front end through retirement are wide (**W**). Although today's front ends cannot support 8-wide fetch (in particular for a variable length ISA such as IA32), the model helps in comparing the benefits of the PARROT microarchitecture to those available through incremental improvements.

The PARROT microarchitecture itself is expressed most naturally through a configuration where a narrow front end is joined to an execution engine capable of executing trace-cache based optimized hot traces. The PARROT configurations are denoted by **TON**, **TOW** and **TOS**, signifying a narrow, wide or split (narrow for cold, wide for hot) cores, respectively. The **T** stands for selective trace-cache and **O** stands for dynamic optimizations.

Configuration Dimensions		Machine width	
		Narrow	*Wide*
PARROT accumulated features	Base	N	W
	Selective TC	TN	TW
	Optimizer	TON	TOW
	Split core	TOS	

Table 3.1 Two-dimensional configurations space

Finally, to assess the significance of the role trace optimizations play in improving performance, two additional configurations are included based on **N** and **W**, including selective trace-caching, but with trace optimizations disabled: **TN** and **TW**, respectively. The microarchitectural properties of all seven configurations are summed up in Table 3.2.

Parameter	N	W	TN / TON	TW / TOW	TOS
Relative core area	1	2	1.3 / 1.4	2.3 / 2.4	3.1
Predictor entries (branch + trace)	4K	8K	2K + 2K	4K + 4K	
FHR length (branch + trace)	16	16	16 + 32	16 + 32	
Icache size	64 KB	64 KB	32 KB	32 KB	
FE pipeline width (cold + hot)	4	8	4 + 4	4 + 8	
ROB entries	100	150	100	150	100 + 150
Sched. Window	32	50	32	50	32 + 50
Exec. Ports	4	8	4	8	4 + 8
EXEC pipeline width	4	8	4	8	4 + 8
Hot filter: entries, threshold	-	-	1K, 24	1K, 8	
Blazing filter: entries, threshold	-	-	1K, 32	1K, 16	
Tcache entries	-	-	128	512	
Max uops in trace	-	-	64		
Optimizer latency	-	-	100 cycles, non pipelined		
L1 Dcache: size, latency	64 KB, 3 cycles				
L2 Ucache: size, latency	2 MB, 9 cycles				
Memory latency	120 cycles				
Line size (I and D)	64 B				
All caches (I, D,T)	8-way associative, LRU				

Table 3.2 μarch settings of different models

3.4. Benchmarks

Our benchmark suite covers a wide range of application traces, 30 or 100 million instructions each. The 44 applications can be classified as follows:

- **SpecInt** 2000: bzip, crafty, eon, gap, gcc, gzip, parser, perlbmk, twolf, vortex, vpr (30M).
- **SpecFP** 2000: ammp, apsi, art, equake, facerec, fma3d, lucas, mesa, sixtrack, swim, wupwise, (30M).
- **Office** / Windows applications from SysMark 2000: excel, office, powerpoint, virusscan, winzip, word (100M).
- **Multimedia**: flash, photoshop (from SysMark-2000, 100M), Dragon, lightwave, quakeIII, 3DsMax (light, anisotropicwheel, raster and geom), two Flask-MPEG4 runs (custom Multimedia traces, 30M).
- **DotNet**: one image, two numerical and two phong runs (100M).

3.5. Metrics

We employ a variety of metrics designated to evaluate and compare different aspects of the simulated models. For the overall processor performance we focus on the IPC, total energy and cubic-MIPS-per-WATT (CMPW) measurements. IPC and total energy are useful for understanding the design tradeoffs assuming the same frequency and same voltage. The CMPW metric is instrumental in quantifying the design tradeoffs and power awareness of a processor assuming energy consumption could be always traded for performance using voltage / frequency scaling [5][34]. (In practice, the applicability range of such tradeoffs is limited by variable technological constraints which are beyond the scope of this research.) We also present some of the most important and illuminating parameters characterizing the PARROT microarchitecture, such as coverage, uop reduction and energy breakdown.

4. Results

This section examines the performance and power awareness of alternative enhancements applied to the reference 4- and 8-wide OOO machines **N** and **W**, respectively. We consider overall performance and power tradeoffs over a variety of configurations and benchmarks, as well as a detailed examination of the contribution of different components to the overall result. The **TOS** conceptual microarchitecture statistics are presented only as a reference for alternative future developments.

The following graphs display the geometrical mean for each group of applications as well as the overall mean for the entire benchmark. Some of the graphs include as well data for three applications that demonstrated the highest improvements. These “killer applications” are flash (multimedia), wupwise (SpecFP) and perlbmk (SpecInt).

4.1. Performance and Power Awareness

We start by examining the impact of the PARROT extensions on the performance of N and W, respectively, as depicted in Figure 4.1. The integration **TW** of a trace cache into the wide machine raises the IPC by an additional 7%, whereas the same extension to the narrow ma-

chine (**TN**) has a negligible 2% performance benefit, mainly since the machine remains balanced for 4-wide execution. However, the PARROT based design that includes gradual optimizations of the hot traces is much better equipped to utilize the available resources. The **TON** model achieves 17% performance improvement over **N** and the full blown **TOW** improves IPC by more than 25% over **W**. It can be observed that the irregular SpecInt applications and the execution-limited multimedia applications benefit the least from the trace-cache alone.

Regarding power, we examine the incremental energy required for achieving the above mentioned performance improvements. Figure 4.2 indicates that all the extensions to the wide machine actually save energy. The reason for that is the vast energy inefficiency of the base wide machine (see Figure 4.5 below). Relative to the narrow machines, only the TW configuration exhibits a significant 12% increase in energy consumption. PARROT style optimizations result in either a negligible 3% increase over **N** or an 18% energy savings over **W**.

The CMPW criterion weighs both the performance gain and the energy loss as indicated in Figure 4.3. Power awareness of the PARROT-based models **TON** and **TOW** improves over the corresponding base machines by 32% and 92%, respectively.

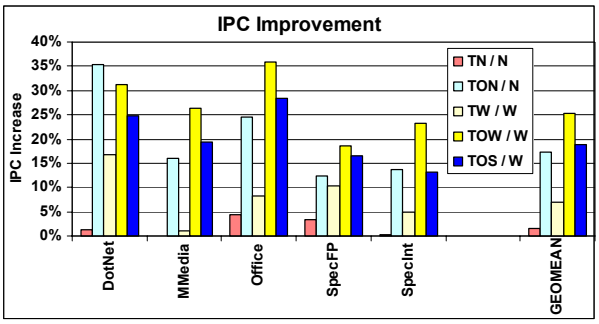

Figure 4.1 IPC improvement over baseline of same width: N or W

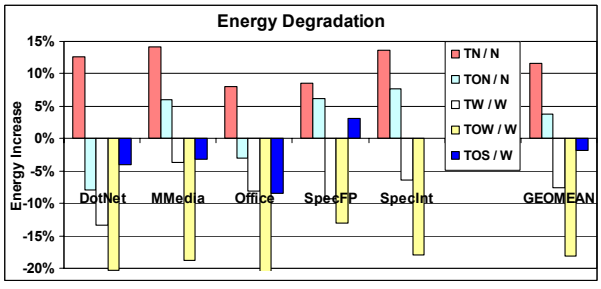

Figure 4.2 Increased energy consumption over baseline

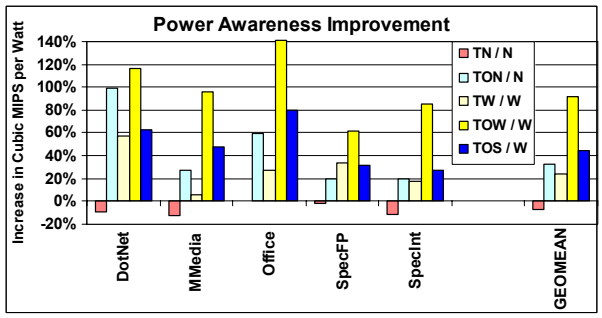

Figure 4.3 Improved power-awareness, over baseline

So far we have presented measurements relative to the base line models. It is of equal interest to consider the trade-offs between the extreme microarchitectural alternatives. To enhance performance on a low power budget, it is most useful to focus on the comparative benefits of the W alternative as compared to TON, which is our proposed solution for a power-limited environment. On the other hand, when designing for a less power-constrained environment, the most aggressive solution (TOW) becomes more interesting. Figure 4.4, Figure 4.5 and Figure 4.6 provide the details. Thus, the PARROT approach gives us best performance at widely differing design points: high end computers that have a good thermal solution and consume significant power, as well as those constrained by a limited power budget.

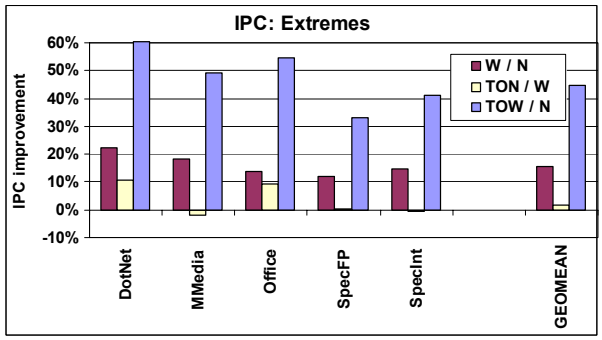

Figure 4.4 IPC improvement across configurations

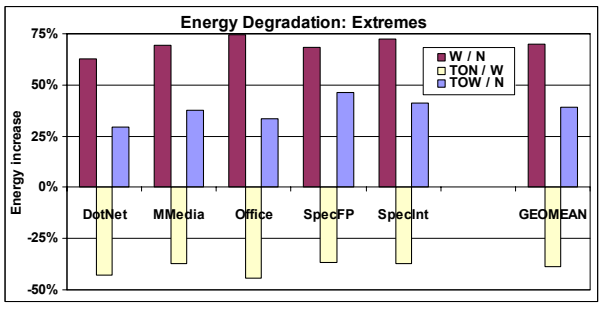

Figure 4.5 Energy degradation across configurations

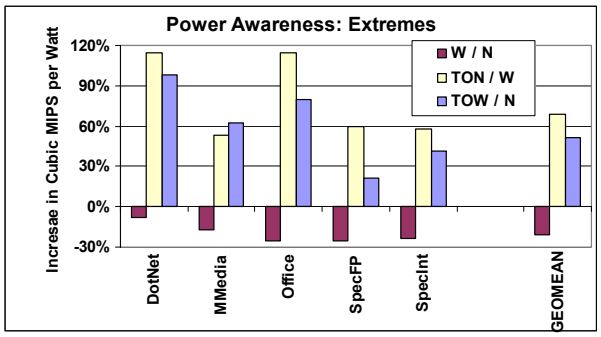

Figure 4.6 Relative CMPW across configurations

First, we observe that widening a machine systematically contributes to performance, but inevitably consumes more energy. The interesting phenomenon is the **TON** configuration which not only slightly outperforms the doubly wide machine **W**, but it does it with significant 39% lower energy consumption. Weighting both metrics into CMPW we can see that the PARROT extensions are 67% better than mere widening. Nevertheless, if a large power budget is available, the combination of both wider machine and PARROT-style extensions exhibited by the TOW configuration can provide an average 45% IPC increase as well as 51% CMPW improvement over the base-line **N**.

4.2. Front-End Capabilities

We demonstrate the better predictability of hot code using misprediction values for the baseline **N** model with a 4K-entries branch predictor versus the PARROT **TON** model with branch and trace predictors, 2K-entries each. Figure 4.7 shows the behavior of the PARROT machine clearly split between the hot code, for which the trace misprediction rate is even smaller than the branch misprediction rate of **N**, as opposed to the cold code, for which the branch misprediction rate is significantly the highest. This split demonstrates the better predictability of the PARROT-constructed hot traces over the cold code.

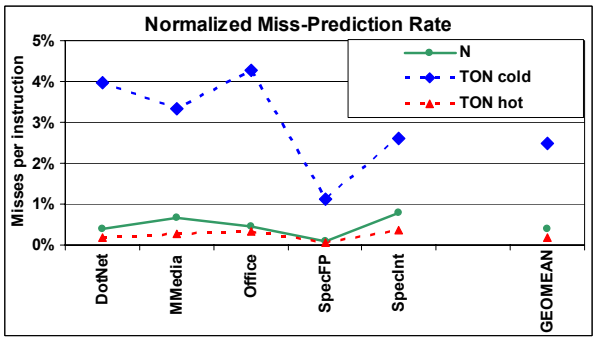

Figure 4.7 Normalized (per instruction) misprediction rates

Coverage, as depicted in Figure 4.8, represents the quality of the trace prediction, selection and filtering mechanisms with respect to the trace-cache size and the benchmark characteristics. The coverage is about 90% for the very regular SPEC-FP applications but around the 60-70% for the control intensive SPEC-INT applications.

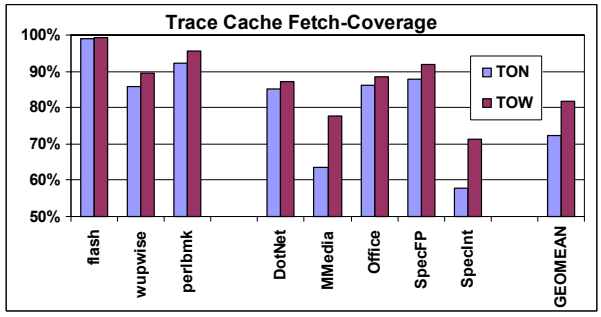

Figure 4.8 Trace-cache hot (fetch) coverage

The impact of a smaller trace cache of the narrow models (128 entries) vs. the wider models (512 entries) is manifested in the large coverage gaps between **TON** and **TOW** exhibited by larger working-set applications such as SPEC-INT or multi-media. Low coverage is particularly harmful for the **TN** configuration in which all the potential performance benefit is due to the trace cache's high fetch-bandwidth. This explains the correlation between the low coverage of SPEC-INT and multi-media applications on the narrow models and the fact that the **TN** model exhibits no IPC improvement for these models (see Figure 4.1 above).

4.3. Optimizer Capabilities

The direct contribution of dynamic optimizations is most significantly reflected in the reduction in the dynamic number of executed uops and the reduction in average trace critical (dependency) path. Reducing the number of uops contributes significantly to both performance gain and reduction in active energy consumption. Shorter critical paths enable better scheduling in terms of higher ILP and shorter execution time. Overall shortening of the application execution time decreases the leakage and idle energy consumption. These contributions of the optimizer are depicted in Figure 4.9 which shows average uop reduction of 19% and average dependency reduction of 8%. Note the relatively higher dependency reduction on the quite complex code of SPEC-INT.

Another important aspect of the microarchitecture is the amount of reuse of the work done by the optimizer. Figure 4.10 depicts the optimization utilization in terms of average number of dynamic execution of optimized traces. The highest reusability is exhibited by the SPEC-FP applications due to the good spatial locality of traces in the trace cache.

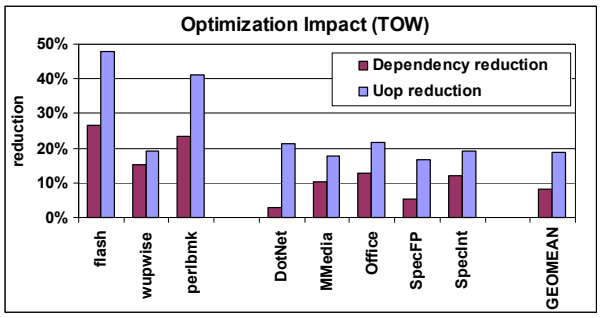

Figure 4.9 Optimizer impact: reducing number of executed uops and simplifying dependencies

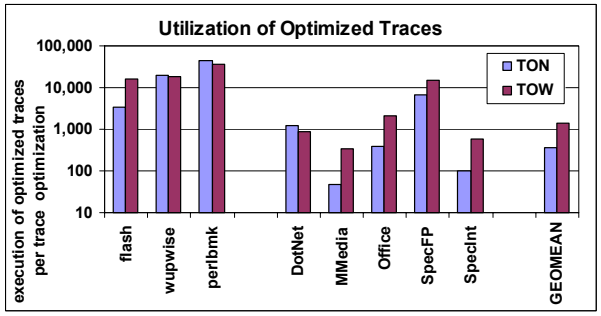

Figure 4.10 Utilization of the optimizer work

4.4. Energy Breakdown

The energy breakdown between the major components of three models is depicted in Figure 4.11. The models include the baseline **N**, the very power-aware narrow PARROT model **TON** and the conceptual, widest PARROT model **TOS**. The breakdown is shown for three applications of quite different characteristics: flash, swim and gcc.

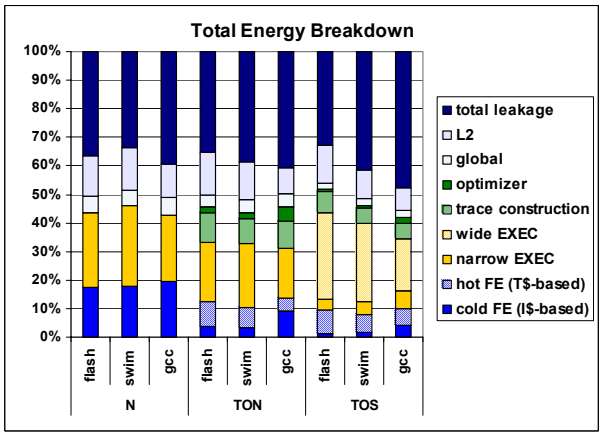

Figure 4.11 Energy breakdown for different models

It is interesting to note the diminishing energy contribution of the front-ends as we move from **N** to **TON** and then to **TOS**. Additionally, on a wider machine such as **TOS** the energy contribution of all execution components increases. Note that total energy required for trace manipulation, including filtering, construction of uops into atomic traces and advanced optimizations is in the order of 10% of the overall energy consumption.

5. Conclusions and Future Work

In this paper we presented the PARROT microarchitectural framework aimed at improving both processor performance and power awareness. The PARROT concept consists of asymmetric decoupling of the processor into subsystems responsible for handling the cold (infrequent) and hot (frequent) portions of the code, thus designing each part according to different power and performance considerations. PARROT based micro-architectures use an instruction cache for the cold code and a decoded trace cache and gradual optimization techniques for executing the hot segments of the code.

We have established clear advantage of the PARROT based approach for designing general-purpose, high-performance power-aware processors. The presented simulation results demonstrate that applying the PARROT concept to a standard, 4-wide, OOO processor yields comparable performance to an 8-wide processor, however, consuming significantly less energy. Applying the PARROT concept to the wider processor achieves significant improvement on both performance and energy.

One major topic for future research is related to split-core micro-architectures. We intend to investigate the potential advantage of such design for establishing even better performance/energy tradeoffs by considering different alternatives for the decoupled split cores.

Acknowledgements

We would like to thank Sanjay Patel, Ronny Ronen and Yiannakis Sazeides for stimulating discussions, encouragement and contributions to earlier research. We thank Lev Finkelstein and Efraim Rotem for their contribution to the power modeling

References

[1] Y. Almog, R. Rosner, N. Schwartz and A. Schmorak, "Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture", in *CGO'04*, March 2004.

[2] V. Bala, E. Duesterwald and S. Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo", TR HPL-1999-78, HP Labs.

[3] [3] M. Bekerman, A. Mendelson and G Sheaffer, "Performance and Hardware Complexity Tradeoffs in Designing Multithreaded Architectures", in *PACT*, pp 24-34, Oct. 1996.

[4] B. Black and J.P. Shen, "Turboscalar: A High Frequency High IPC Microarchitecture", in *ISCA27*, June 2000.

[5] D.M. Brooks et al, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors", *IEEE Micro*, 20(6):36-44, Nov./Dec. 2000.

[6] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: a Framework for Architectural-level Power Analysis and Optimizations", in *ISCA27*, 83-94, June 2000,

[7] G. Cai, C.H. Lim and W.R. Daasch, "Thermal-Scheduling For Ultra Low Power Mobile Microprocessor", in Proc. *WCED'02*, 2002.

[8] A. Dhodapkar, C. Lim, G. Cai and R. Daasch, "TEM^{2}P^{2}EST: A Thermal Enabled Multi–Model Power/Performance ESTimator", in *PACS* Workshop, held in conjunction with ASPLOS, 2000.

[9] K. Ebcioglu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", in *ISCA24*, pp. 26-37, 1997.

[10] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel and S.S. Lumetta, "Performance Characterization of a Hardware Mechanism for Dynamic Optimization", *MICRO34*, Dec. 2001.

[11] M Franklin and G.S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism", in ISCA19, 1992.

[12] D. Friendly, S. Patel and Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", in *MICRO31*, Nov. 1998.

[13] M. Gschwind, E.R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, "Dynamic and Transparent Binary Translation", in *IEEE Computer Magazine* 33(3), pp. 54-59, 2000.

[14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel, "The Microarchitecture of the Pentium® 4 Processor", in *Intel Technology Journal*, 2001.

[15] Q. Jacobson, E. Rotenberg and J.E. Smith, "Path-Based Next Trace Prediction", in *MICRO30*, 1997.

[16] S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, and R. Ronen, "eXtended Block Cache", in *HPCA6*, Jan. 2000.

[17] O. Kosyakovsky, A. Mendelson and A. Kolodny, "The Use of Profile-based Trace Classification for Improving the Power and Performance of Trace Cache Systems", in *4th FDDO*, Austin, Dec. 2001.

[18] M.S. Lam and R.P. Wilson, "Limits of Control Flow on Parallelism", in Proc. *19th ISCA*, pp. 46 -57, May 1992.

[19] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, "Effective Compiler Support for Predicated Execution using the Hyperblock", in *MICRO25*, 1992.

[20] S. Melvin and Y Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA", in *Intern. Journal of Parallel Prog*., 23(3) pp 221-243, Jun. 1995

[21] M.C. Merten, A.R. Trick, C.N. George, J. Gyllenhaal, and W.W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization", in *ISCA26*, 1999.

[22] M.C. Merten, A.R. Trick, E. M. Nystrom, R.D. Barnes and W. Mwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots", in *ISCA27*, May 2000.

[23] R. Nair and M.E. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups", in *ISCA24*, pp. 13-25, 1997.

[24] A. Parikh, M. Kandemir, N. Vijaykrishnan and M.J. Irwin, "VLIW Scheduling for Energy and Performance" in *Proc. IEEE Workshop on VLIW*, pp. 111-117. April 2001.

[25] S. Patel and S. Lumetta, "rePlay: A Hardware Framework for Dynamic Optimization", in *IEEE Trans. on Computers*, 50(6), pp 590-608, June 2001

[26] S. Patel, T. Tung, S Bose and M. Crum, "Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions", in *MICRO33*, 2000.

[27] A. Peleg and U. Weiser. "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", U.S. Patent 5,381,533, Jan. 1995.

[28] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", in *Journal of ILP*, vol. 1, Oct. 1999.

[29] R. Rosner, A. Mendelson and R. Ronen, "Filtering Techniques to Improve Trace-Cache Efficiency", in *PACT'01*, Sept. 2001.

[30] R. Rosner, M. Moffie, Y. Sazeides and R. Ronen, "Selecting Long Atomic Traces for High Coverage", in *ICS'03*, pp. 2-11, 2003.

[31] E. Rotenberg, S. Bennett and J. Smith, "A Trace Cache Microarchitecture and Evaluation", in *IEEE Trans. on Computers*, 48(2), pp 111–120, Feb. 1999

[32] B. Solomon, R. Ronen, D. Orenstien, Y. Almog and A. Mendelson "Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA", in *ISLPED'01*, Aug. 2001.

[33] B. Slechta et al., "Dynamic Optimizations of Micro-Operations", in *HPCA9*, Feb. 2003.

[34] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski and P.G. Emma, "Optimizing Pipelines for Power and Performance", *MICRO35*, 2002.